CS 319 – Applied Programming

Lecture # 19

Wednesday, December 09, 2015

FALL 2015

FAST – NUCES, Chiniot-Faisalabad Campus

Mian Rizwan Ul Haq

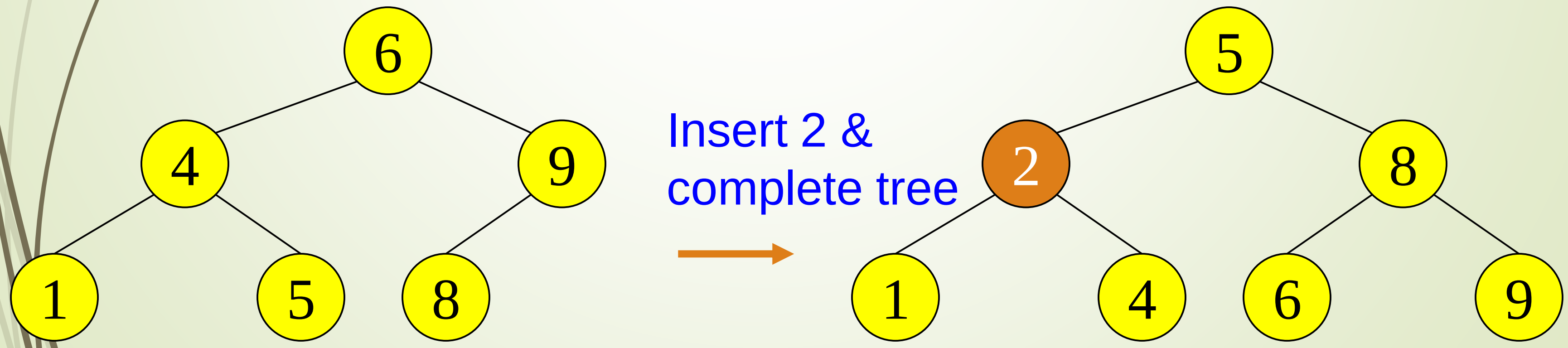
rizwan.haq@nu.edu.pk

(Adelson-Velsky and Landis)

-

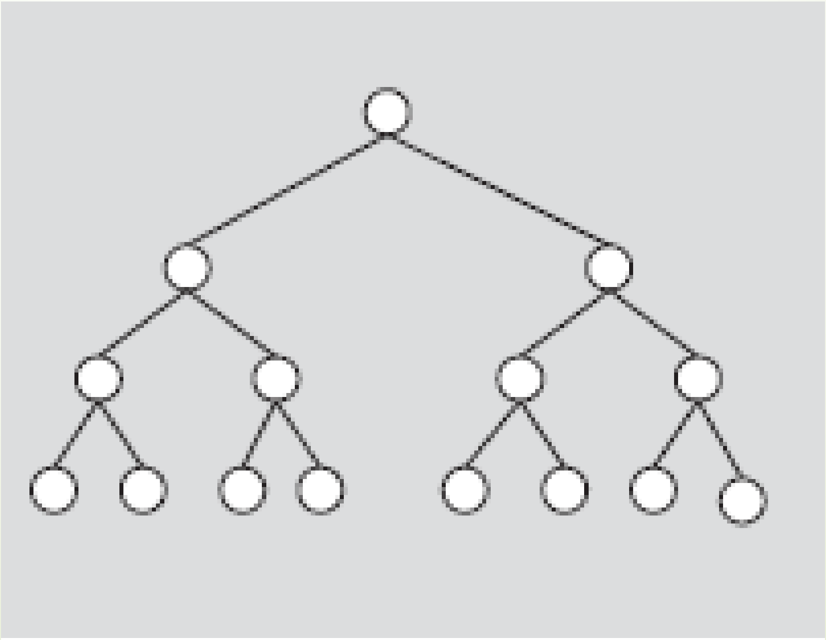
Complete tree

- Want a complete tree after every operation
  - tree is full except possibly in the lower right
- This is expensive
  - For example, insert 2 in the tree on the left and then rebuild as a complete tree



Perfectly Balanced Tree

- A perfectly balanced tree
  - The heights of the left and right subtrees of the root are equal.
  - The left and right subtrees of the root are perfectly balanced binary trees

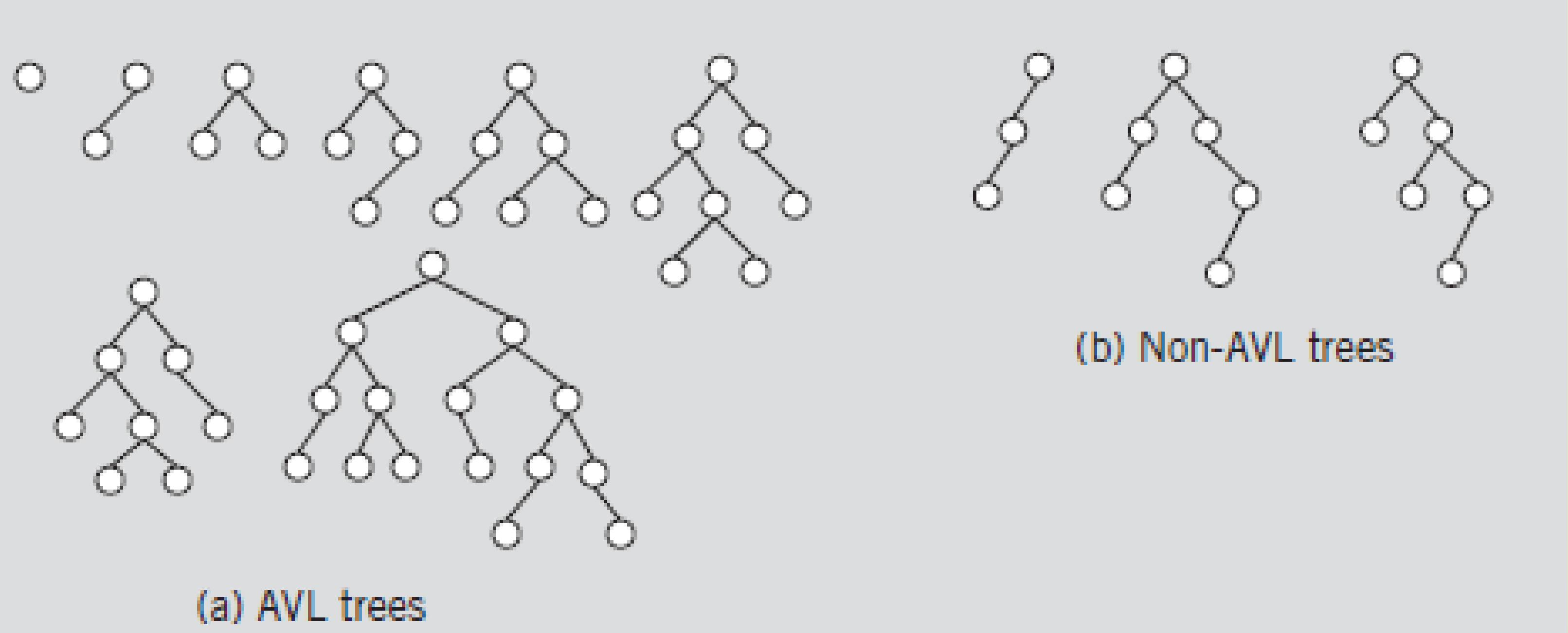


Perfectly Balanced Tree

- If ***T*** is a tree of height ***h*** then it can be proved that number of nodes in the tree ***T*** is $\mathbf{2^h - 1}$
- What should be the number of elements in a data set of height ***h*** to construct a perfectly balanced tree from it???

Height-Balanced tree (AVL Tree)

- A binary search tree such that (Not perfectly balanced)
  - The heights of the left and right subtrees of the root differ by at most 1
  - The left and right subtrees of the root are AVL trees



Proposition for AVL

- Proposition: Let T be an AVL tree and x be a node in T and x_l and x_h be the height of left and right sub tree respectively. Then $|x_h - x_l| \leq 1$, where $|x_h - x_l|$ denotes the absolute value of $x_h - x_l$.
- Let x be a node in the AVL tree T.
  - If $x_l > x_h$, we say that x is left high. In this case, $x_l = x_h + 1$
  - If $x_l = x_h$, we say that x is equal high
  - If $x_h > x_l$, we say that x is right high. In this case, $x_h = x_l + 1$

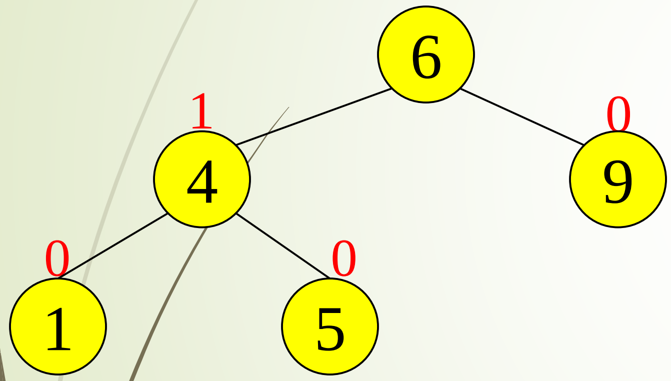
AVL Trees

- The height of the left subtree minus the height of the right subtree of a node is called the **balance of the node(Balancing Factor)**.
- Balance factor of a node x is written as *bf(x)*
- $bf(x) = x_l - x_h$
- Let x be a node in tree T then
  - If x is left high, *bf(x)* = 1.
  - If x is equal high, *bf(x)* = 0.
  - If x is right high, *bf(x)* = -1.
- The height of an empty tree is defined to be 0.

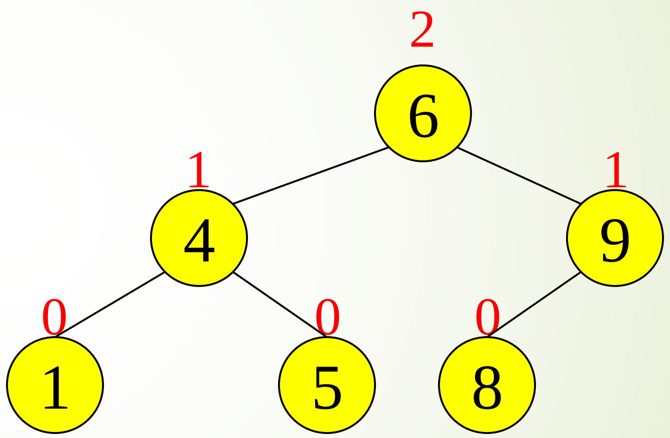
Node Heights

Tree A (AVL)

Height = 2 BF = 1-0 = 1



Tree B (AVL)



height of node = h

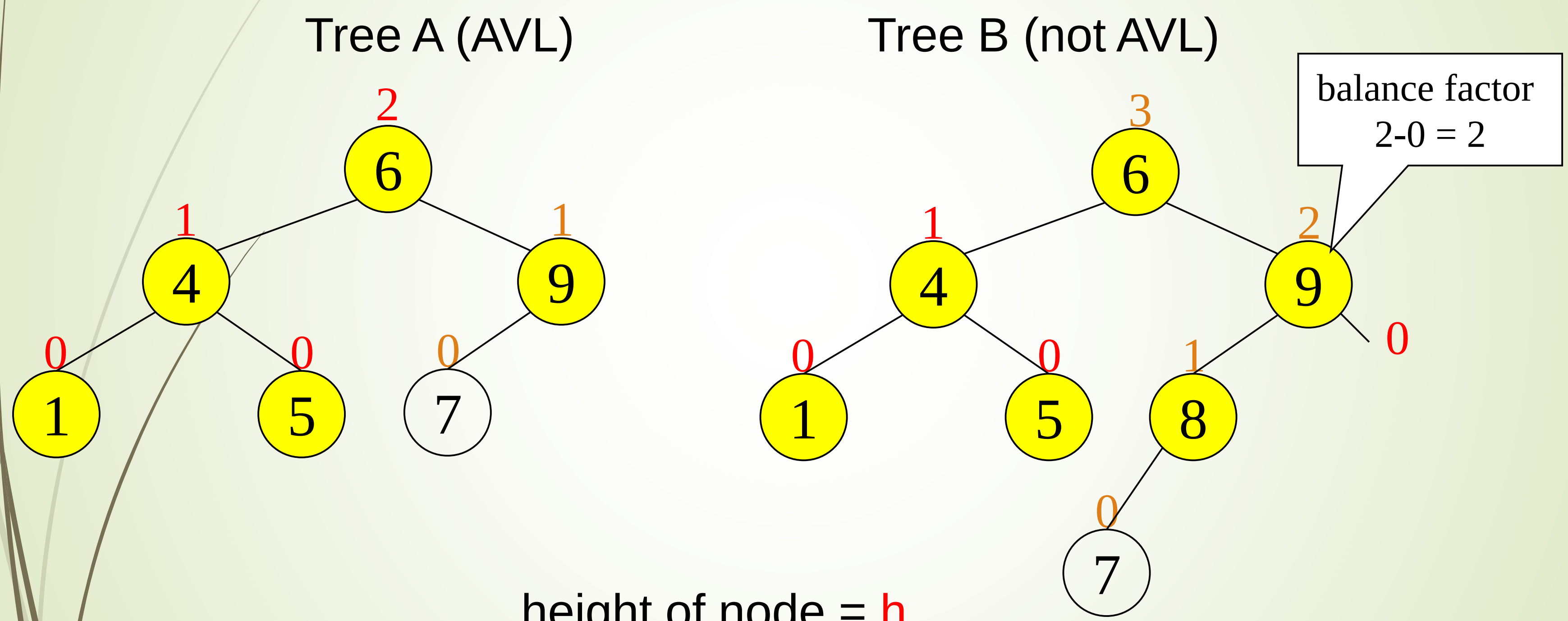
balance factor = h_{left}-h_{right}

empty height = 0

AVL Trees

- Given an AVL tree, if insertions or deletions are performed, the AVL tree **may not** remain height balanced.

Node Heights after Insert 7



height of node = h
balance factor = h_{left}-h_{right}
empty height = 0

AVL Tree node

```
template<class elemType>
struct AVLNode
{
    elemType info;
    int bfactor; //balance factor
    AVLNode<elemType> *llink;
    AVLNode<elemType> *rlink;
};
```

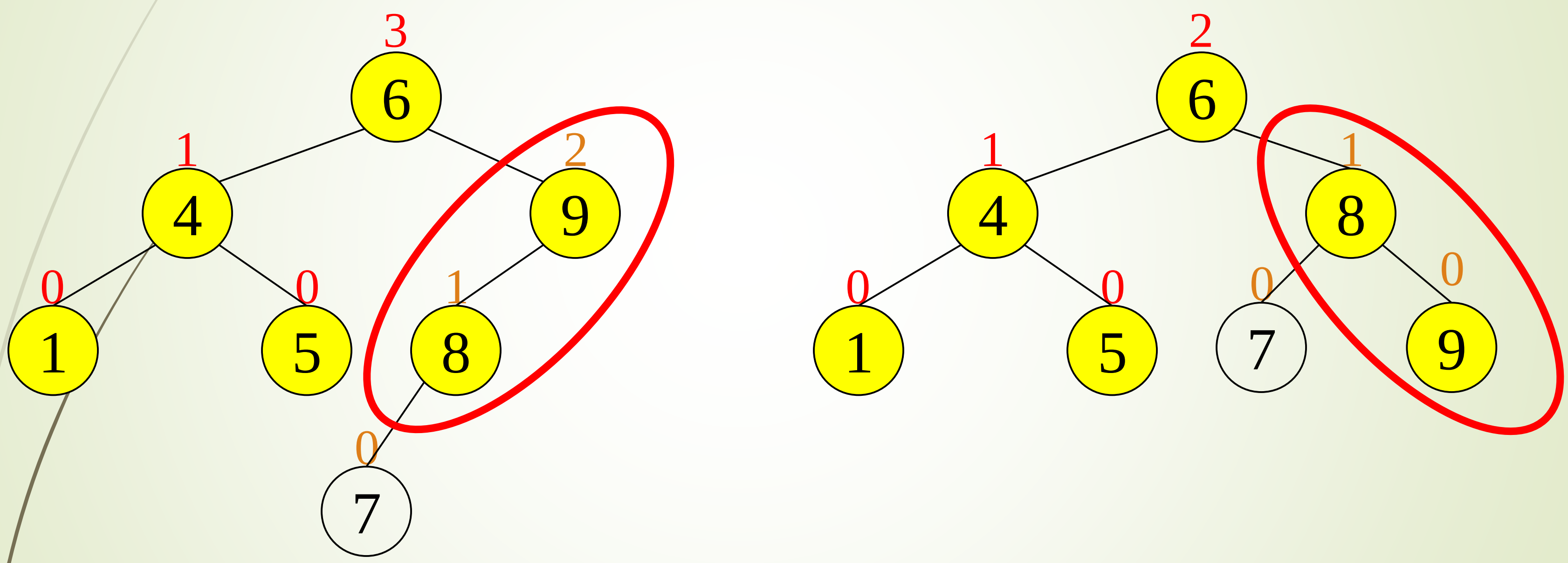
AVL Trees

- To maintain the height balanced property of the AVL tree after insertion or deletion, it is necessary to perform a transformation on the tree so that
  - The inorder traversal of the transformed tree is the same as for the original tree (i.e., the new tree remains a binary search tree)
  - The tree after transformation is height balanced

Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - Follow the path up to the root, find the first node (i.e., deepest) whose new balance violates the AVL condition. Call this node ***a***
  - If ***a*** new balance factor (the difference h_l-h_h) is 2 or –2, adjust tree by rotation around the node

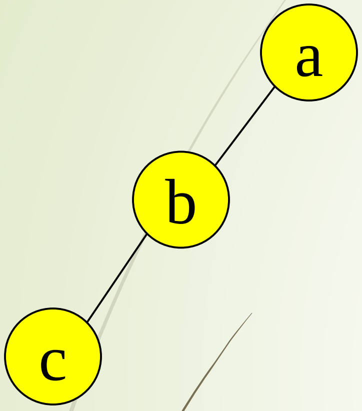
Single Rotation in an AVL Tree

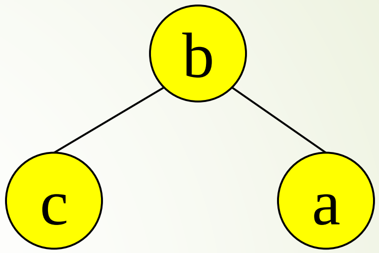


Left Rotation (LL) in an AVL Tree

- *Node **b** will takes place of node **a*** (it will become new Root).
- *Node **a** will become root of left sub-tree(or left child of new Root)*
- *If there was any left child of node **b**, it will become right child of node **a** now*

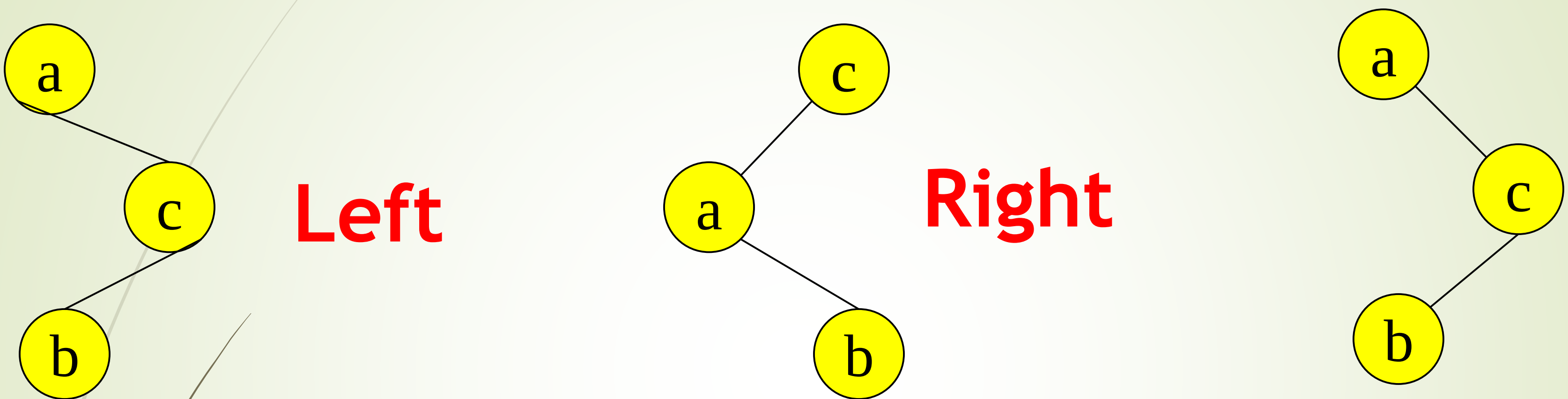
Right Rotation (RR) in an AVL Tree





- *Node **b** will takes place of node **a** (it becomes root now)*
- *Node **a** will become root of the right sub-tree (or right child of new root)*
- *If there was any right child of node **b**, it will become left child of node **a** now.*

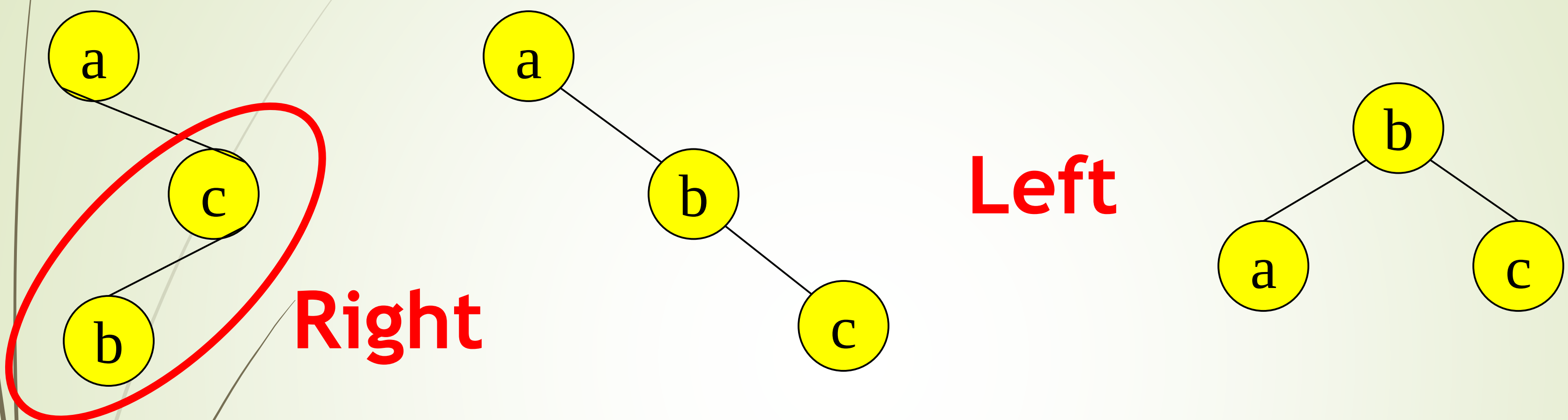
Single Rotation may be Insufficient



- **c** *becomes the new root.*
- **a** *takes ownership of* **c's** *left child as its right child, in this case,* **b**.
- **c** *takes ownership of* **a** *as its left child.*

- **a** *becomes the new root.*
- **c** *takes ownership of* **a's** *right child as its left child,* **b**.
- **a** *takes ownership of* **c** *as its right child.*

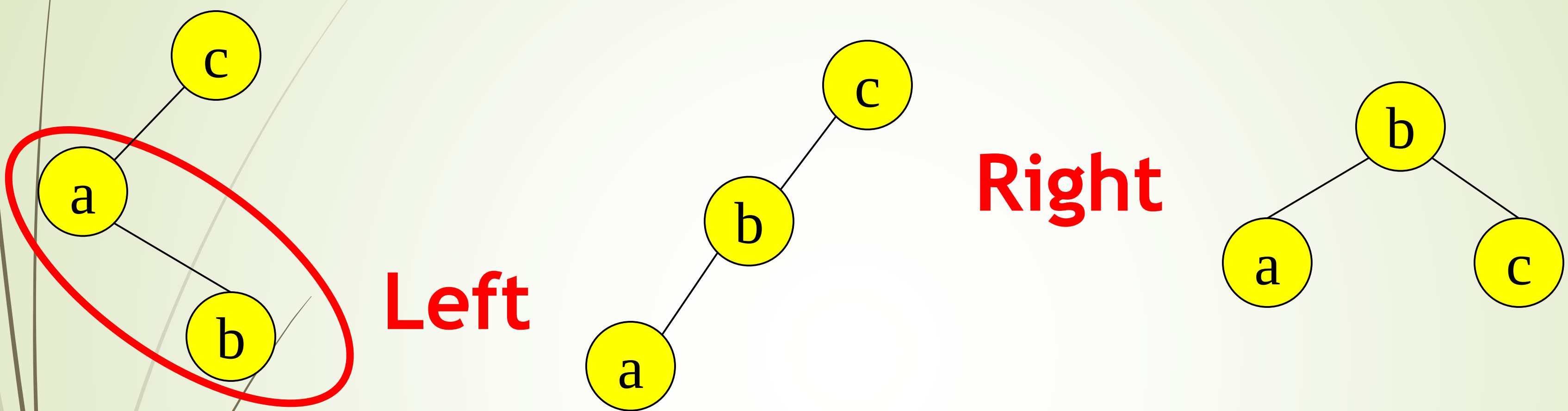
Left-Right Rotation (LR) or Double Left



- *perform **a** right rotation on the right subtree.*

- ***b** becomes the new root.*
- ***a** takes ownership of **b's** left child as its right child, in this case null*
- ***b** takes ownership of **a** as its left child*

Right-Left Rotation (RL) or Double Right



- *perform* ***a*** *left rotation on the left subtree.*

- ***b*** *becomes the new root.*
- ***c*** *takes ownership of* ***b's*** *right child as its left child, in this case null.*
- ***b*** *takes ownership of* ***c*** *as its right child.*

How and when to rotate?

- If the BF values of A and its immediate node(B) are both positive: Make a single right rotation(RR)
- If the BF values of A and its immediate node(B) are both negative: Make a single left rotation(LL)
- If the BF value of A is positive and its immediate node(B) has a negative BF value: Make a double right rotation(RL)
- If the BF value of A is negative and its immediate node(B) has a positive BF value: Make double left rotation(LR)

How and when to rotate?

- Double Left Rotation
  - Make Single Right Rotation at node B
  - Make Single Left Rotation at node A

- Double Right Rotation
  - Make a Single Left Rotation at node B
  - Make a single Right Rotation at node A

Working Example

- Example: 40 30 20 10 5 15 65 33 31

Exercise - I

- Insert the following data into the empty AVL tree, “Mar, May, Nov, Aug, Apr, Jan, Dec, July, Feb, June, Oct, Sept”

Reading Material

- Schaum's Outlines: Chapter # 7
- D. S. Malik: Chapter # 11
- Nell Dale: Chapter # 8
- Allen Weiss: Chapter # 4
- Tenebaum: Chapter # 5